Achieving Fast Fully Homomorphic Encryption with Graph Reductions

Walden Yan and Sanath Govindarajan
Mentor: William Moses
What is fully homomorphic encryption?

- Support arbitrary computation on encrypted data

Alice (Sender)

Encryption
\[ \mu \rightarrow \text{Enc(}\mu) \]

Decryption
\[ \text{Dec(F}'\text{(Enc(}\mu))) = F(\mu) \]

Eve (Untrusted Receiver)

Computation
\[ \text{Enc(}\mu) \rightarrow \text{F}'\text{(Enc(}\mu)) \]
Potential Applications

- We can send tasks off to someone with a more powerful computer or a better algorithm without having to worry about data leaks
  - Filtering email and messages
  - Processing medical data
  - Processing financial data
  - National security
But it is slow

\[
\begin{align*}
\text{Enc}(\mu) & \downarrow \\
F_1(\text{Enc}(\mu)) & \downarrow \\
F_2(F_1(\text{Enc}(\mu))) & \downarrow \\
\vdots & \\
\text{Func}'(\text{Enc}(\mu)) & \\
\end{align*}
\]
Our Contribution
Program $F(\mu)$

Encrypted Data Corpus $\text{Enc}(\mu_1), \text{Enc}(\mu_2), \ldots$

Optimized Encrypted Program $F': \text{Enc}(\mu) \rightarrow \text{Enc}(F(\mu))$

Encrypted Outputs $\text{Enc}(F(\mu_1)), \text{Enc}(F(\mu_2)), \ldots$

FHE Pipeline

- Function Graph IR
- Reduce Graph
- Optimized Graph
- Lower to Scheme
- Halide IR
- Schedule & Compile
- “Library” Function
Function Graph IR
Function Graphs

- DAG of binary operations

3-bit addition
Measuring Graph Efficiency

- Benchmark individual binary operations in the FHE scheme

- In the worst case, the time it takes to run the graph is the sum of the time it takes to run each individual operation
  - Could be faster due to parallelism or schedule optimizations

- Theoretically, any scheme can be used

<table>
<thead>
<tr>
<th>Operation</th>
<th>NOT</th>
<th>AND</th>
<th>XOR</th>
<th>XNOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime (relative to NOT)</td>
<td>1</td>
<td>18.75</td>
<td>38.71</td>
<td>35.72</td>
</tr>
</tbody>
</table>
Graph Reductions

3-bit multiplication

3-bit multiplication (reduced)
Eliminating Constants and Double NOTs

- Any binary operation taking a constant as an input can be expressed solely in terms of the other input
  - $\text{XOR}(A, 1) = \text{NOT}(A)$

- $\text{NOT}(\text{NOT}(A)) = A$
Optimizing 2-Input Graphs

- Given a graph with two input nodes and some desired outputs, find the best graph to compute those outputs

- 2 inputs $\Rightarrow$ 4 possible sets of inputs $\Rightarrow$ 16 possible functions $\Rightarrow$ 65536 unique sets of outputs
- Run a DP algorithm to find all the optimal graphs and cache them in a table
- Use the table to find the optimal graph for any situation
Generic Graph Reduction

For all pairs of nodes $u$ and $v$:

- Define the subgraph $S$ as all nodes that can be calculated from only $u$ and $v$
  - Approximate with DFS
- Consider node $w$ in $S$ interesting if $w$ is used outside of $S$ or if $w$ is an output of the original graph
- Run the 2-input graph algorithm with interesting nodes as desired outputs
- Replace the $S$ with the ideal subgraph

Repeat until graph cannot be reduced further
Two-Node Reduction on Full Adder
Additional Reduction Methods

- Three-node reduction

- Find exact subgraph $S$ by running every possible set of inputs and analyzing patterns in node values

- Flag “important” input nodes (ex. sign bits)
  - Try creating separate graphs for when the bit is 0 and when it is 1, then combine with MUX
Scheduling and Compiling
Our FHE Scheme

- **GSW 2013**: leveled fully homomorphic encryption scheme based on LWE [1]
  - Ciphers are matrices, operations are matrix addition & multiplication
  - Requirement for leveled FHE: plaintext $\mu \in \{0,1\}$ at all times
- **NOT** ($\mu$) = 1 - $\mu$
- **AND** ($\mu_1 \mu_2$) = $\mu_1 \ast u_2$
- **XOR** ($\mu_1 \mu_2$) = AND ($\mu_1 (\lnot \mu_2)$) + AND ($\lnot \mu_1 \mu_2$)
- **XNOR** ($\mu_1 \mu_2$) = AND ($\mu_1 \mu_2$) + AND ($\lnot \mu_1 \lnot \mu_2$)
  - Graph optimizations take differing costs of operations into account
- Since all encrypted gates are matrix operations, we can use a tensor processing compiler to generate fast code

Implementing Fast FHE Operations

- We use Halide, a high-performance image and tensor processing compiler
- Algorithms are separated from schedules
  - Implement FHE operator once
  - Halide can schedule/compile for many architectures (caching differences, CPU/GPU, etc)
- Easy parallelization by design (no side effects, etc)
Homomorphic AND in Halide

//Simplified for ease

Halide::Func AND(Halide::Func f1, Halide::Func f2, int matSize) {
    Halide::Var x, y;
    Halide::RDom r(0, matSize);
    Halide::Func multiply_temp;

    multiply_temp(x, y) = Halide::Expr((int64_t)0);
    multiply_temp(x, y) += f1(x, r) * f2(r, y); //modular sum in practice

    return Flatten(multiply_temp);
}
How We Generate Pipelines

Input

Input

Input

ImageParam

ImageParam

ImageParam

Public params (modulus, size)

Primitives

Output

Output

Output

Realization

Halide compiles this to return a callable function pointer
Compiling a function graph

```cpp
vector<ImageParam> inputPlaceholders(2 * num_bits);

for (int i = 0; i < inputPlaceholders.size(); i++) {
    inputPlaceholders[i] = ImageParam(Int(64), 2);
}

Pipeline hpipe = pipelineGen(some_function, inputPlaceholders, N, q); // pipeline ready to be scheduled

// scheduling here, or use the auto-scheduler

hpipe.compile_jit(); // or compile_to_c or any other supported language
Realization rel = hpipe.realize(N, N, Target(), params); // ready to be decrypted
```
A “Dynamic” Library

- Given an FHE program, see if we’ve already compiled it, if so return/call it
- Otherwise compile a pipeline to compute the operation
  - Moderately slow, but can be reused
- Can either JIT or ahead-of-time compile depending on use case
API
Creating Graphs: Building From Scratch

```python
function_graph fg(3);  // 3 input bits
int node1 = fg.addNode(fg.getInput(0), fg.getInput(1), AND_OP);
int node2 = fg.addNode(fg.getInput(2), node1, OR_OP);
fg.defineOutput(0, node2);
reduce(fg);  // also has optional flags
```
Creating Graphs: Using Standard Operations

function_graph fg;
var x(fg, 0, 5); // inputs 0...4
var y(fg, 5, 5); // inputs 5...9
var z(fg, 10, 5); // inputs 10...15
var res = (x + y) / z;
function_graph opGraph = res.realize();
Results
FHE Scheme Benchmarking

\[ O((n \log q)^3) \]
Optimization Benchmarking

0.27 ms reduction

3.5 x reduction
Optimization Benchmarking

- Division: 3.5 x reduction
- Multiplication: 2.8 x reduction
Optimization Benchmarking

ReLU: \( \frac{|x| + x}{2} \)

17.5 x reduction
Conclusion

● A pipeline that speeds up the running of programs with fully homomorphic encryption
  ○ Internal representation that can be optimized with graph reductions
  ○ Scheduling and compiling homomorphic programs with Halide
● A basic API for easy use of the pipeline
● Demonstrated significant speedups compared to using bare fully homomorphic encryption
Future Work

- Adding heuristics to better handle larger function graphs
- Allowing function graphs to incorporate lower level FHE operations
- Adding new primitive gates (ex. MUX)
- Incorporate RLWE to allow faster arithmetic operations
- Improving the API
Acknowledgements

- Our parents
- Our mentor, William Moses
- Dr. Slava Gerovitch
- Professor Srini Devadas